Requirements Eng (2006) 11: 123–137
DOI 10.1007/s00766-005-0024-3

ORIGINAL ARTICLE

Sebastian Uchitel · Robert Chatley
Jeff Kramer · Jeff Magee

Goal and scenario validation: a fluent combination

Received: 30 June 2004 / Accepted: 27 April 2005 / Published online: 20 December 2005

Abstract Scenarios and goals are effective techniques for requirements definition. Goals are objectives that a system has to meet. They are elaborated into a structure that decomposes declarative goals into goals that can be formulated in terms of events and can be controlled or monitored by the system. Scenarios are operational examples of system usage. Validation of goals and scenarios is essential in order to ensure that they represent what stakeholders actually want. Rather than validating scenarios and goals separately, possibly driving the elaboration of one through the validation of another, this paper exploits the relationship between goals and scenarios. The aim is to provide effective graphical animations as a means of supporting such a validation. The relation between scenarios and goals is established by means of fluents that describe how events of the operational description change the state of the basic propositions from which goals are expressed. Graphical animations are specified in terms of fluents and driven by a behaviour model synthesised from the operational scenarios. In addition, goal model checking over operational scenarios is provided to guide animations through goal violation traces.

1 Introduction

Scenario-based specifications, such as message sequence charts (MSCs) [1] and UML sequence diagrams [2], are popular as part of requirements specifications. Scenarios describe how system components, the environment and users work concurrently and interact in order to provide system level functionality. Their simplicity and intuitive graphical representation facilitate stakeholder involvement and make them popular for documenting intended system behaviour. Each scenario is a partial description which, when combined with all other scenarios, contributes to the overall system description. However, scenarios are operational descriptions, and consequently leave the required properties of the intended system implicit. Although scenarios provide a common ground on which goals can be elicited, discussed, and elaborated, they are not requirements in their own right.

Goals [3] are properties that are expected to be achieved by the system [3, 4], where system refers to the software and its environment [5]. Goals are declarative statements and as such complement the operational nature of scenarios. In the context of goal oriented requirements engineering [3], goals are refined into subgoals creating goal graphs that show how high-level goals are realised by lower level ones. Low-level goals can be operationalised and assigned to specific components. In this way, an operational model can be inferred from the goal graph. Summarising, goals can serve as a framework for eliciting and elaborating operational descriptions of the required system.

The relationship between scenarios and goals has been extensively researched. More specifically, focus has mainly been on how scenarios can aid the elaboration of goals and vice versa. In [6], van Lamsveerde discusses this aspect of the relationship extensively and uses the dotted elements of Fig. 1 to depict how goal- and scenario-based RE can be integrated (arrows depict data dependencies). Scenarios may prompt the elicitation of underlying goals or can be used for goal inference (Fig. 1, right-to-left arrow). They can provide examples of how goals can be realised or be witnesses of violations of these goals (Fig. 1, left-to-right arrow). Goals may drive the elaboration of new scenarios (Fig. 1, left-to-right arrow). However, less attention has been given to how scenarios and goals can be used together to facilitate requirements elicitation, elaboration and validation. In this paper we focus specifically on validation.

S. Uchitel (✉) · R. Chatley · J. Kramer · J. Magee
Department of Computing, Imperial College London,
London, UK
E-mail: s.uchitel@doc.ic.ac.uk
E-mail: rbc@doc.ic.ac.uk
E-mail: jk@doc.ic.ac.uk
E-mail: jnm@doc.ic.ac.uk

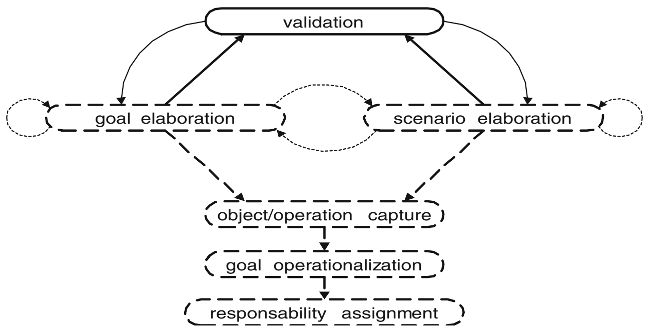


Fig. 1 Interweaving goal- and scenario-based RE (*Dashed elements* taken from [6])

Validation is a key requirements activity. Requirements engineers must not only elicit and document scenarios and goals, but also validate that these are indeed what stakeholders want [7]. Animation is an effective validation technique, and we show that combining goals with scenarios can result in effective animations that not only support scenario and goal validation, but can also prompt their elaboration.

In its simplest form, animation means allowing stakeholders to step through sequences of events dictated by some behaviour model. More sophisticated animations allow stakeholders to visualise some aspect of the system state as they step through the alternative events. However, this is not straightforward to implement if behaviour has been described in the form of scenarios, where the notion of state is implicit. Which system states should be inferred from what are essentially sequences of events? One possibility could be to assume that each event in the scenario leads to a new system state. However, this is too naïve.

To present a meaningful animation to a stakeholder, the choice of states to visualise must relate to the concerns of the stakeholder participating in the animation. System goals can be the source for establishing a meaningful partition of system state, because they capture the reasons why the system behaves as described in the scenarios.

We believe that animations of system behaviour should be driven by scenarios and displayed from the perspective of the stakeholder participating in the animation and in a way that relates to the stakeholder's goals. For example, the intermediate states that the system may go through while performing user authentication could be considered irrelevant from the perspective of a user accessing email on a web-based email client. However, given a privacy goal on email access, partitioning the system state space into those where the user has been authenticated and logged on, from those in which authentication has failed, is very relevant.

To perform such animations, a relation between scenarios and goals must be established. We build on van Lamsveerde's approach, in which goals are successively AND–OR refined (Fig. 1, left circle arrow) into requisites. Requisites are goals that can be formulated in terms of predicates whose truth value depends on events that can be controlled or monitored by an individual agent. However, in our approach we do not require goals to be refined to such a low level. We allow goals that are not requisites to be related to operational scenarios. We use fluent linear temporal logic (FLTL) formulas [8] to express goals that can be formulated in terms of predicates whose values depend on system events. These FLTL formulas have as basic elements fluents, which are central to establishing the relation between goals and scenarios.

Fluents are abstractions of system state specified in terms of the occurrence of events such as those that appear in operational scenarios. Miller and Shanahan [9] informally define (propositional) fluents as follows: "Fluents (time-varying properties of the world) are true at particular time points, if they have been initiated by an event occurrence at some earlier time point, and not terminated by another event occurrence in the meantime. Similarly, a fluent is false at a particular time point if it has been previously terminated and not initiated in the meantime."

Fluents are also central to specifying animations. The same fluents that are used to express system goals are used to construct visualisation rules that are used by an animator to build graphical views of the system. This results in intuitive, simple visualisation specifications, and also in animations that support validation of goals and scenarios used in conjunction. By having stakeholders animate system behaviour viewed through abstract states defined in terms of fluents, not only are the scenarios validated as they drive the animation but also confidence in the validity of the system goals is gained. As a consequence of relating the goals to the scenarios through fluents, the scenarios can be model checked for goal satisfaction.

The toolset that we have developed to support our approach uses standard web browsers and HTTP and is particularly well suited for the animation of web-based applications.

The paper is organised as follows: Section 2 presents background on MSCs, labelled transition systems (LTSs) and model synthesis. We also introduce a small example to illustrate our approach. Section 3 introduces fluents and goals and Section 4 gives an overview to our animation approach. Section 5 details how visualisations are specified using fluents. In Section 6 we describe how our approach can prompt goal and scenario elaboration. Section 7 discusses a model checking of goals and Section 8 explains the multi-user extensions. Section 9 gives an account of our experience using the approach and the paper concludes with a discussion and related work.

2 Message sequence charts and behaviour models

The notation we use for documenting scenarios is a syntactic subset of the MSC standard of the Interna-

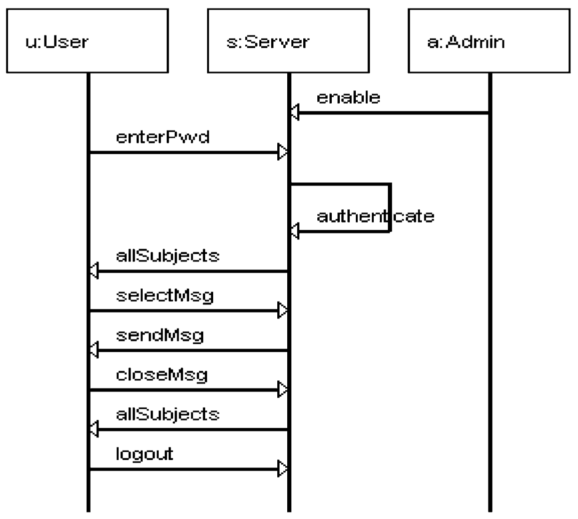


Fig. 2 User reads email scenario

tional Telecommunication Union [1] and of UML 2.0 sequence diagrams [2]. For a detailed and formal description of the language refer to [10]. A scenario-based specification consists of several basic MSCs and one high-level MSC (hMSC). A basic MSC (bMSC) describes a finite interaction between a set of components (see Fig. 2). Each vertical line, called an instance, represents a component. Each horizontal arrow represents a synchronous message, its source on one instance corresponds to a message output and its target on a different instance corresponds to a message input. Arrows that have the same component as source and target are internal component events. A bMSC defines a partial ordering of messages, which in turn defines a set of sequences of message labels (called traces) that are all the possible orderings of the partial order of messages. In Fig. 2 only one system trace is defined: *<enable, enterPwd, authenticate, allSubjects, selectMsg, sendMsg, closeMsg, allSubjects, logout>*.

A hMSC allows the sequential composition of bMSCs. It is a directed graph where nodes represent bMSCs and edges indicate their possible continuations. The hMSC shows how the system can evolve from one bMSC to another. In other words, an edge in the graph indicates that the source bMSC can be followed by the target bMSC (note that the semantics of hMSCs uses weak sequential composition, for a detailed discussion on the semantics we use refer to [10]). Figure 3 shows the hMSC for a simple model of a web-based email system. Once a user has logged in, they can view messages. Alternatively, they can fail to log in correctly (possibly repeatedly). In addition, the administrator may disable the user at any point. hMSCs also have an initial node (the init box in Fig. 3). The behaviour of an MSC specification is given by a set of sequences of message labels: those determined by composing sequentially the bMSCs of any maximal path in the hMSC, where a maximal path is a path that cannot be extended further. Note that this corresponds to the adoption of weak sequential composition, which is the standard interpretation of hMSCs [1].

We use LTSs [11] to model the behaviour of communicating components in a concurrent system. A LTS (Fig. 4) is a state transition system where transitions are labelled. Transition labels model the messages components send and receive. A trace of an LTS *P* is a sequence of observable events that *P* can perform starting at its initial state. In addition, we use an operation on LTS called parallel composition (based on the || composition operator used in CSP [12]) to model the system that results from composing components such that they execute asynchronously but synchronise on all shared

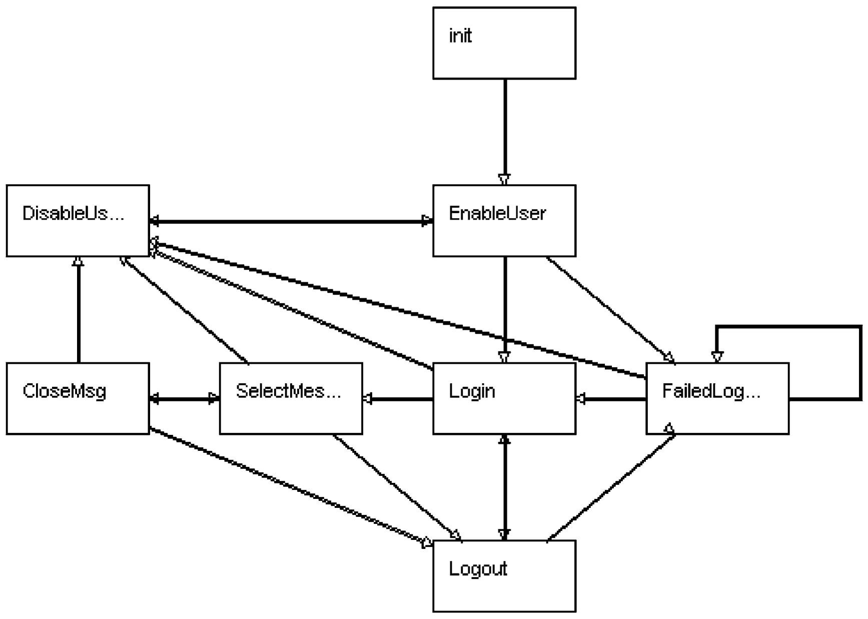


Fig. 3 High-level message sequence charts (MSCs) for Web-mail system

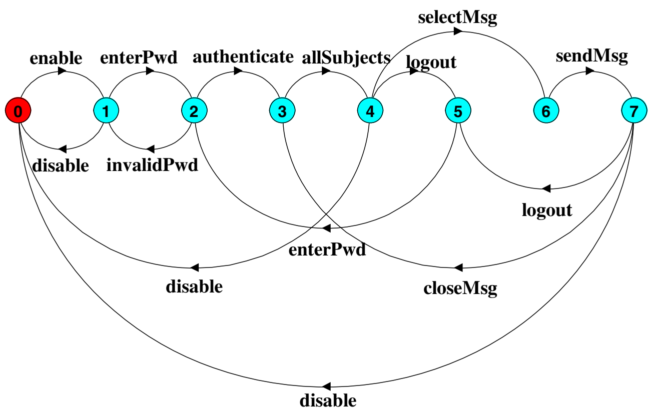


Fig. 4 Synthesised LTS for Server component

message labels. In other words, message labels are interpreted as handshaking communication between components. For a detailed explanation refer to [13].

We use an LTS synthesis technique to automate the construction of behaviour models from MSCs [14]. We construct one model for each component that appears in the MSC specification. (Figure 4 shows the LTS of the Server component from Fig. 2.) The LTS alphabet is the set of messages the component sends and receives in the MSC specification.

Once a LTS has been synthesised for each component, the parallel composition of all LTSs yields what we call a minimal architecture model of the MSC specification. This means that it is the smallest model with respect to trace inclusion that preserves the component structure and interfaces of the MSC specification and that exhibits all the traces specified in the MSC specification. Note that minimality is not enough to guarantee that the architecture model will not provide unspecified behaviours: implied scenarios can arise due to mismatches between the component structure and the traces specified in the MSC specification. For a detailed explanation of architecture models, synthesis and implied scenarios refer to [10].

3 Goals and fluents

We use goals in the spirit of van Lamsveerde's goal-oriented requirements engineering approach, KAOS (see [3] for an overview). Goals are considered to be objectives that a system should meet where system refers to the software-to-be and its environment. Hence, in this paper goals subsume inner-requirements [15] and software-system-to-be properties. Goals are declarative statements as opposed to operational descriptions meaning that they address WHAT the system should do rather than HOW. High-level goals are AND-refined into sets of sub-goals, meaning that the satisfaction of sub-goals is a sufficient condition for the satisfaction of the higher-level goals. By AND-refinement high-level goals can be decomposed into goals (called requisites) that can be formulated in terms of states variables whose values depend on events that are controllable or monitorable by some individual component (see [16] for a discussion on monitored and controlled events). At this stage, Lamsveerde's approach advocates assigning requisites to components and elaborating how the component can realise the requisite through a series of operations. This entails that there is a strong relation between component operations and the predicates on states from which requisites are formulated. In our event-based models, it is natural to formulate requisites from propositions that are predicates on the occurrence of events. Thus, we propose using "fluent" propositions to naturally formulate goals in terms of states controllable by a set of components.

From [8], we define a fluent Fl by a pair of sets, a set of initiating actions I_{Fl} and a set of terminating actions T_{Fl}: $Fl \equiv \langle I_{Fl}, T_{Fl} \rangle$ where $I_{Fl}, T_{Fl} \subset Act$ and $I_{Fl} \cap T_{Fl} = \emptyset$, and Act is the set of all actions. In addition, a fluent Fl may initially be true or false at time zero as denoted by the attribute $Initially_{Fl}$. The set of atomic propositions from which FLTL (the linear temporal logic of fluents) formulas are built is the set of fluents Φ. Therefore, an interpretation in FLTL is an infinite word over 2^{Φ}, which assigns to each time instant the set of fluents that hold at that time instant. An infinite trace $< a_0 a_1 a_2 \ldots >$ over Act also defines an FLTL interpretation $< f_0 f_1 f_2 \ldots >$ over 2^{Φ} as follows:

$\forall i \in N \forall Fl \in \Phi : Fl \in f_i$ **iff** either of the following holds

- $Initially_{Fl} \wedge (\forall k \in N \cdot 0 \le k \le i, a_k \notin T_{Fl})$
- $\exists j \in N : ((j \le i) \wedge (a_j \in I_{Fl}) \wedge (\forall k \in N \cdot j < k \le i, a_k \notin T_{Fl}))$

In other words, a fluent holds at a time instant if and only if it holds initially or some initiating action has occurred, and in both cases, no terminating action has yet occurred. Using the syntax of the LTSA [17] we can specify the following fluents that relate to the simple email system:

```
fluent LoggedIn = <authenticate, {logout, disable}>
fluent Registered = <enable, disable>
fluent ReadingMsg = <sendMsg, closeMsg>
```

The `LoggedIn` fluent specifies that for a user to be in the logged in state, that user must have previously been authenticated by an `authenticate` action and that the user must not have logged out or been disabled by the administrator. The `Registered` fluent specifies that a user is registered from the point that user is enabled by the administrator until disabled. The `ReadingMsg` fluent specifies the state in which a user can read a message (Note that the default initial value for fluents is false). Given these fluents, we can specify firstly, the system goal that a logged in user must always be registered and secondly, that a user must always be logged in to read a message. In the following, `[]` is the temporal always operator, $\rightarrow$ implication and && conjunction. Using this notation, we express two goals for the WebMail system as follows:

```
assert LegalAccess =[](LoggedIn -> Registered)
assert PrivateRead =[](ReadingMsg -> LoggedIn)
```

These two goals can be related to a higher-level goal concerned with the security of the system. In doing goal decomposition, this higher-level goal is refined by the conjunction of LegalAccess and PrivateRead. Thus, having formalised the lower-level goals, the higher-level goal is specified by:

```
assert SecureEmailAccess = (LegalAccess && PrivateRead)
```

In terms of van Lamsveerde's goal-oriented requirements engineering approach, KAOS [3], the high-level goal graph for the WebMail system is given in Fig. 5.

The fluents involved in PrivateRead include events from several different agents. For instance, logout is an event shared by user and server, while disable is an interaction between the administrator and the server. In the approach described in this paper, we do not require goals to be formulated in terms of states controllable by an individual component (i.e. requisites), but potentially by a set of components that may interact in order to realise the goal. These components and their interactions are described in the operational scenarios.

Note that assertions in FLTL differ from state-based assertions (e.g. OCL [2]) in that they do not refer to values of state variables that have been defined within the behaviour specification. In FLTL there are no state variables as all behaviour is described in terms of sequences of events. In addition, note that fluents differ from modes used in SCR [18] in two ways. Firstly, fluents are not part of the behaviour specification, hence they do not constrain the behaviour specification formalism nor the way the formalism is used to describe system behaviour (For instance, LegalAccess does not need to be a concept explicit within the behaviour specification). Secondly, fluents can be defined over events of different system components, while modes must refer to events of one machine.

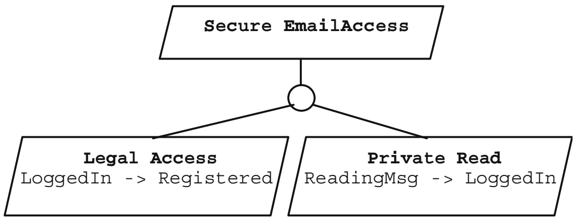


Fig. 5 Simple goal graph for WebMail

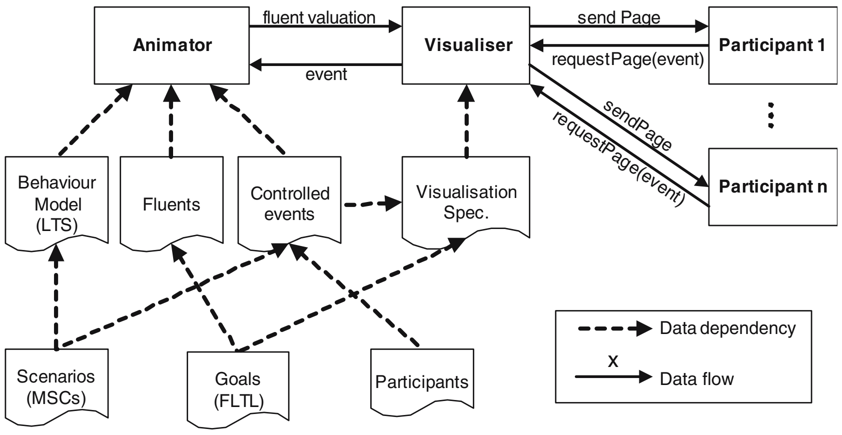
4 Animation

In this section, we give an overview of the model we adopt for animation. In Section 5, we described the details of how systems states captured by fluents are rendered as web pages.

Animation is performed by three components: an animator, a visualiser and a participant (top of Fig. 6). In fact, there can be several participants, each of whom interacts through a standard web browser; however, to simplify presentation we defer the explanation of multi-user animations to Section 7. Our conceptual animation model relies on inputs (bottom of Fig. 6): scenarios in the form of MSCs, a goal graph that has been refined to goals that can be formulated in terms of fluents, and the roles that participants are to play in the animation.

The animator component uses a behaviour model in the form of a LTS that is the result of the LTS synthesis (Section 2 and [10]) from the given scenarios. The animator uses the behaviour model to react to events controlled by the animation participants. The animator also has access to the definitions of the fluents used in specifying goals. It keeps track of the value of fluents during an animation and forwards these values to the visualiser, which in turn uses them to construct visualisations of system state, based on these values. Finally, the animator is informed as to which transitions of the model correspond to events that are to be controlled by the participant. The choice of these controlled events depends mainly on the role stake-

Fig. 6 Conceptual animation model



holders participating in the animation are to have. For instance, if a participant is to play the role of a user of the WebMail system described previously, the set of controlled events is: `{enterPwd, selectMsg, closeMsg}`.

The visualiser component requires a visualisation specification. This specification is a mapping from fluents to visual elements. When the component needs to produce a visualisation of the system state, it will compose all visual elements that correspond to fluents that are true in that state. The visual elements also include active elements (such as buttons and hyperlinks) that are related to events that are controllable by the participant. These active elements allow participants to trigger the occurrence of the events they control. Animation proceeds as follows:

START:

1. Animator runs model until a state is reached in which all outgoing actions are labelled by controlled actions. The values of fluents are kept up to date as actions are executed. Fluent values passed to Visualiser.
2. Visualiser accepts HTTP request and returns HTML rendering of page using fluent values.LOOP:

3. Visualiser accepts HTTP request with selected event parameter. Event passed to Animator.
4. If the event does not correspond to an outgoing transition then ERROR, otherwise, model state advanced as in 1 above and updated fluent values passed to Visualiser.
5. Visualiser returns HTML rendering of page according to current fluent values.Note that animation only renders stable states, where a stable state is one in which all outgoing transitions are labelled by controlled actions. This form of maximal progress is widely adopted in modelling reactive systems and is, for example, consistent with Statechart semantics in which all microsteps, grouped into a macrostep, are executed before a transition to the next state occurs. It is also consistent with the semantics of languages such as Lustre [19] and Esterel [20].

The tool that we have developed to support our approach is an extension of the labelled transition systems analyser (LTSA) tool [17]. LTSA serves as the animator component of our model, while a specially developed plug-in implements the Visualiser component. This plug-in has a small web server within it, and that can serve pages to the different participants' standard web browsers.

5 Specifying visualisation using fluents

The visualisation specification relates the system state with a web page that is presented to participants to convey that state. The specification thus consists of rules that map state to visual elements. Rather than tying these visualisation rules to concrete system states (e.g. the states of the labelled transition system synthesised from scenarios), we map fluent expressions which characterise abstract states to visual elements.

Note that fluent expressions define abstract states that crosscut the concrete states of the behaviour model. This is because the truth value of a fluent does not depend on the concrete state the behaviour model is in at a given point, rather it depends on the history of events that led to the concrete state. This feature decouples behaviour and goal modelling, providing an abstract mechanism for linking the two.

Using abstract states that are relevant to the participant and that relate to system goals is more effective in supporting scenario and goal validation since many concrete system states in a behavioural model are not directly meaningful to a stakeholder in the context of a specific animation.

Fluent expressions are constituted by fluent propositions used in system goals, expressed in FLTL. Examples of these were presented in Section 3. We associate fluent expressions with visualisation elements by means of *showwhen* rules. In the context of the LTSA tool, these rules are encoded in XML. In the following, we present an abstract syntax in which bracketing by tags is represented by indentation. Tags are shown in bold.

```
showwhen
    not LoggedIn
display
    <!— HTML for graphics, username, password boxes -->
    table tr
      td input type="text" name="userid"
      td input type="password" name="pwd"
      td button enterPwd
```

The above visualisation rule specifies that the HTML in the display section is included in the returned web page when the fluent `LoggedIn` does not hold. The display section of a rule uses HTML extended with some additional constructs which are rewritten by the Visualiser when constructing a web page. One of these additional constructs is **button**—this specifies a controlled action in the model that is returned as an attribute of the HTTP request when the button, that the construct causes to be displayed on the web page, is pressed. In other words, pressing the button on the web page allows a model transition labelled with the action—in this case `enterPwd`—to occur.

Instead of the **button** we can use hyperlinks to control actions. For instance, the following can be used to display a message subject that, if clicked on, would trigger the action `selectMsg`. Images can be associated to the link in the same manner.

```
link
  action selectMsg
  content "Your diploma has arrived"
```

Typically, a visualisation specification will have many *showwhen* rules. When the visualiser receives the truth values of all fluents from the animator, it builds a web page by aggregating all of the HTML fragments in *showwhen* rules whose expressions evaluate to true. In addition, the specification may include an HTML header and footer to be included on every generated page. This helps to provide a consistent look and feel to the pages of the visualisation.

In order to enrich the visualisation, we allow the possibility of displaying data previously entered or selected by the user. For instance, when logging in, the user supplies their username. On subsequent screens, we can use this information to add a greeting at the top of the page using the following rule.

```
showwhen
   loggedIn
display
   <!-- greet user -->
   "Hello"
   value name="userid"
   "welcome to webmail."
```

The **value** tag is another extension to HTML that is rewritten by the visualiser and replaced with the value that the user entered when logging in. The name "userid" matches the name of one of the **input** elements in the previous rule.

We also support the addition of behavioural constraints based on data input by the user. For example, we may specify that the `authenticate` event will only be performed when the username and password that the user typed match particular values. This provides the participant with a better experience of the system than if they were, say, authenticated or not based only on a non-deterministic choice, which is the behaviour specified in the scenarios for this example.

```
action authenticate
conditions
  and
   equal key="userid" value="demo"
   equal key="pwd" value="demo"
```

6 Exploiting inconsistency in visualisation specifications

The fact that user controlled events are made accessible to the participant by means of *showwhen* rules may lead to inconsistency. These inconsistencies may simply signal a trivial mistake in the visualisation specification; however, more importantly, they can also prompt the elaboration of scenarios and goals

There are two manifestations of an inconsistent visualisation specification. The first is when a controlled event is made available to a participant at a certain point during the animation, but the underlying behaviour model does not allow that controlled event to occur. In these cases, if the participant triggers the event, the animator cannot react to it as no scenario describing the appropriate behaviour was given.

This type of inconsistency occurs when there is a mismatch between visualisation and behaviour. It may be the case that the visualisation criteria are correct (that it is reasonable to allow the participant to trigger the event in the current abstract state) and that the scenario being animated was not considered in the original set of scenarios. Hence the inconsistency signals an incompleteness of the scenario specification. On the other hand, it may be the case that the abstract state has been incorrectly defined, and consequently some visual elements are being displayed inappropriately. Incorrect definition of the abstract state can be a result of incorrect fluent expressions, or incorrect definition of the initiating and finalising events for fluents. In both the cases, because fluents and fluent expressions are extracted from goals, goal elaboration may be required. Finally, the inconsistency may signal that the goals are not being satisfied by the operational behaviour of the system; hence revision of either goals or scenarios is required.

The second manifestation of an inconsistent visualisation specification is when a controlled event is not made available to the participant at a state when the event is possible in the underlying behaviour model. This means that the participant is being denied the possibility of animating certain system behaviours. These inconsistencies may indicate the existence of superfluous scenarios or, as before, a problem with the fluents and fluent expressions defining abstract states. In both the cases, the inconsistency may prompt the elaboration of scenarios and goals.

The animation tool recognises these inconsistencies and informs a participant of when they occur. If the participant clicks on a controlled event that is not enabled in the underlying behaviour model, the visualiser will return an error message to them—step 4 of the outline animation algorithm of Section 4. In addition, when the visualiser component builds a page for the participant, it checks if there are any enabled controlled events in the current state of the model, that are members of the controlled set of actions for that participant, and for which the page has no active elements. If this is so, it adds default buttons for them to the page. A message is also displayed to highlight the inconsistency.

7 Model checking goals

Although animation techniques are effective to support validation and elaboration, they rely on participants exploring system behaviour sufficiently thoroughly as to cover relevant situations. A complementary approach is to use model checking techniques to find traces of particular interest and to use them to direct the animation.

In this way, the animation can lead participants through uses of the system that need special consideration.

Model checking of goals expressed in FLTL can be done using the FLTL2Büchi plug-in for our tool LTSA [8]. Given a FLTL formula ϕ and a LTS M for which we wish to check if ϕ holds, the plug-in first produces a Büchi automaton that recognises all traces that violate ϕ. Secondly, the plug-in can check whether the intersection of the languages accepted by the Büchi automata and M is empty. A trace in the intersection is a violation of ϕ, and if such a trace exists the model checking algorithm returns it. In order to optimise model checking, the plug-in will produce a LTS instead of a Büchi automaton if the produced ϕ is a safety property. In this case, the generated LTS has an error state which when composed with M is guaranteed to be reachable if and only if ϕ does not hold in M. The LTS generated by the FLTL2Büchi plug-in for property `PrivateRead` (see Section 3) is depicted in Figs. 7 and 8. In this LTS, all sequences of actions starting at the initial state (labelled 0) that lead to the error state (labelled -1) are examples of violations of the `PrivateRead` property. However, not all of these sequences correspond to traces that can be exhibited by the WebMail model.

In the context of the WebMail example, we wish to check whether the three goals asserted in Section 3 hold in a model that captures the combined behaviour of user, server and administrator. To do so we must first compose the behaviour models synthesised in Section 2 using parallel composition (composite processes in FSP are declared by prefixing their name with `||`):

```
||WebMail = (User || Server || Admin).
```

If `PrivateRead` is now checked against `WebMail` the model checker reports a violation. The tool can produce output in two different formats. Figure 9 shows the violating trace in textual format annotated with the fluents that are true at each point in the trace. An alternative representation can be generated in the form of an MSC (Fig. 10). The trace shows how the administrator enables the user and the user then enters the password that is successfully authenticated. At this point, the fluent `LoggedIn` becomes true. The user then receives the subjects of emails in the inbox, selects a message to read and receives the content of the message.

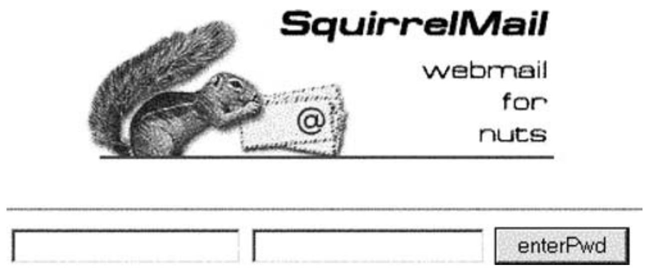


Fig. 7 Visualisation of not `LoggedIn`

At this point, the fluent `ReadingMsg` also becomes true. The next event to occur is the disabling of the user by the administrator. This leads to the `LoggedIn` fluent becoming false and property `PrivateRead` being violated.

Note that to further support feedback on violations to goals, it is possible to use the fluent-based animation approach presented in this paper to produce an animation of the trace returned by the model checker.

Interestingly, the violation highlights an important mismatch between system goals and system architecture. The goal requires users to be logged in if they are reading a message. Hence when a user is disabled while reading a message a violation occurs. There are several ways to attempt a solution to this problem. The first is to incorporate some mechanisms that will delete the content of the message from the user's screen as soon as the administrator disables the user. However, because of the pull nature of web browser–server communication, it is impossible for the server to push (i.e. change) the state of the browser unless the browser requests a new page. Hence, the goal `PrivateRead` is too strong for the architecture of the system. As a consequence, a second possibility is to weaken the goal `PrivateRead` to state that the content of a message cannot be received if the user is not logged on. This goal can be expressed as follows in FLTL and can be proven to hold in `WebMail` by the model checker.

Note that `WeakPrivateRead` uses an event label as a fluent. An event label defines a fluent that becomes

```
assert WeakPrivateRead=[](sendMsg->Registered)
```

true with the occurrence of that event and becomes false with any other event.

8 Multi-participant animations

Based on our experience with scenarios and animations, we have recognised the benefits of supporting multi-participant animations. These animations allow stakeholders to explore how the behaviour of system entities affects each other. Multi-participant animations are particularly useful in concurrent and distributed systems, and systems that can be used concurrently by multiple users.

To support multi-participant animations, visualisation specification provides the notion of *role*. Hence, controllable events and visualisation rules are defined on a role basis, and each animation participant is given a role during the animation.

In this way, multiple participants can control different sets of events and have completely different visualisations of system state. Each visualisation is more akin to the fluents, and hence the goals, that are relevant to each role.

Multi-participant animations do, however, introduce some additional behaviour. The reason for this is the choice of a decoupled architecture of our tool: Web

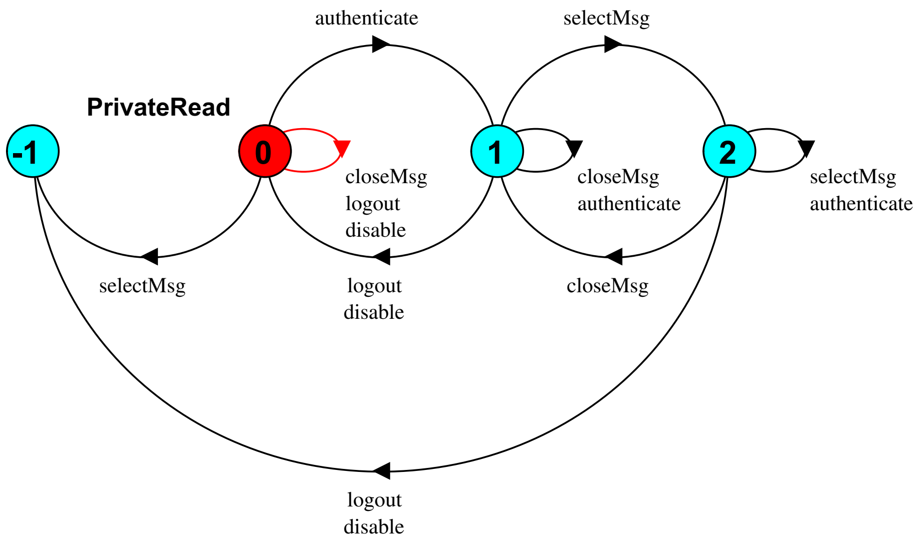


Fig. 8 Property LTS for `PrivateRead`

```
Trace to property violation in PrivateRead
   enable
   enterPwd
   authenticate     LoggedIn
   allSubjects      LoggedIn
   selectMessage    LoggedIn
   sendMessage      ReadingMsg && LoggedIn
   Disable          ReadingMsg
```

Fig. 9 Trace violating `PrivateRead` goal annotated with values of fluents

browsers used by participants can only request web pages from the Visualiser. Hence, the visualiser cannot inform participants of any change of state if the browser is not refreshed. This is crucial in a multi-participant animation. If one user is visualising the state through their browser, and meanwhile another user has triggered an event and hence a change of state, the first user will not see the change of state. The consequence of this is that the first user may choose to trigger an event that was enabled in the original state, but is no longer enabled.

The situation described above, is exactly what happens in web-based applications, and makes the architecture we have adopted particularly well suited for animating them. Consequently, we have extended our tool to cope with these situations and to provide appropriate feedback when they arise.

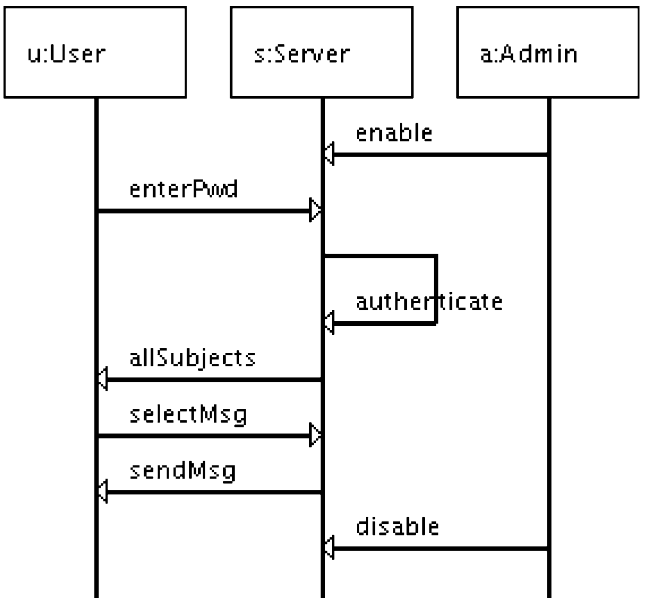


Fig. 10 MSC representation of violation to `PrivateRead` goal

9 Experience

The techniques described above have been used to create an early prototype of an application called eSuite, developed by the Greek software company LogicDIS. The eSuite application provides a layer on top of an enterprise resource planning (ERP) system that enables users to interact with the system via a web interface. Versions have been developed to work with all of the most popular ERP systems. Typical uses of the system include stock control and the placing and monitoring of orders for products. LogicDIS are in the process of developing a new version of eSuite, to improve functionality and usability over the first released version. In order to save on development effort, they wished to validate their designs for the new system with their users before beginning the development phase.

Working with LogicDIS developers, we created a scenario specification that detailed the intended behaviour of a particular part of the system: the order insertion procedure. The overall goal that the user hopes to achieve using this part of the system is that an order is placed once they receive an instruction from a customer (perhaps by telephone). The customers do not enter orders into the system directly (they are not the users), but liase with a salesperson. A particular salesperson may be responsible for taking orders from several customers from different companies.

We used the Milestone Refinement Pattern [21] repeatedly to decompose this goal and determine a number of intermediate states that need to be achieved in order to achieve the overall goal. These were as follows: to have selected a company, selected a customer, completed payment and delivery details (the order header), and completed the details of the order. More formally (where <> is the Eventually temporal operator):

A set of fluents were derived from these subgoals and, based on these fluents, a set of visualisation rules. These were expressed in the XML format required by the visualisation tool. For example, the customerSelected fluent was specified as follows:

```
fluent="customerSelected"
  initiates
    action selectCust
  terminates
     action goToSelectCust
     action insert
```

The XML fragment shows the name of the fluent, and the set of initiating actions (just `selectCust`) and terminating actions (`goToSelectCust` and `insert`). The initiating action is fairly self-explanatory; once the user has performed the `selectCust` action they have selected a customer and the fluent becomes true. The reasons for including the actions in the terminates set are less obvious.

The insert action is the last action the user performs in the order insertion scenario. This action is included in the terminates set of all of the fluents that describe whether a particular subgoal has been achieved, so that all of the fluents are "reset" before the scenario is performed again. If this is not done, then it is only possible to run through the scenario once with the user, as on trying to repeat it, the visualiser will not display any of the intermediate states of the order insertion process, as the fluents mark them as having already been completed.

The `goToSelectCust` action was added in to the terminates set for this fluent as the design was refined. During validation with users, a need was expressed to be able to go back to a particular stage in the insertion process, to correct an entry. Scenarios were added to the MSC specification to allow links back to earlier parts of the process to be followed. With this jumping back, we want to redo certain parts of the process, for instance to select a different customer. In order for the visualiser to redisplay the customer selection screen, we need to "undo" the achievement of one of the subgoals. To do this, we include `goToSelectCust` in the set of actions that terminates `customerSelected`.

In total the model of the order insertion process comprises 15 scenarios, 12 fluents and 19 showwhen rules. The full sets of fluents and showwhen rules are given in Tables 1 and 2. Below we show the showwhen rule (again in our compressed format—the actual specification is full XML) for displaying the list of companies for the user to select from. The link constructs are translated to normal hyperlinks on the web page, and the data fields are used to record information that can be displayed on later pages (in this example, we record the name of the selected company so that it can be used throughout the order).

To try and capture the look-and-feel of the eSuite application, buttons and images from LogicDIS's graphic designer (mostly taken from the original version of eSuite) were included. Special headers and footers were added to the visualisation specification to allow a consistent look to be given across all pages. To allow these to be displayed on every page, we defined the never fluent, which is negated to form a fluent expression that is always true. We did not define an always fluent directly, as the visualiser initialises all fluents as being false. In places, buttons were customised with graphics by placing an image in the button's content field rather than text. The visualiser translates this to the appropriate HTML.

The screenshot in Fig. 11, 12 and 13 shows the view that the user sees towards the end of the process when they are assembling the products that make up the order. At this point they have successfully selected a company and customer, and completed the order header, but have not yet specified the details of the order. This screen gives a good impression of how graphics can be included to allow users to walk through scenarios in the context of an interface that approximates that of the finished product.

The photograph in Fig. 14 shows us working with users. The users were given a view to interact with and asked to perform certain tasks. This initiated discussion as to how well the system supported their achieving their

Table 1 Fluents used in the eSuite simulation

```
fluent companySelected  = < selectComp ,{ insert , gotoSelectCompany } >
fluent customerSelected = < selectCust , { insert , gotoSelectCustomer}>
fluent searched =         < search , { insert , selectCust } >
fluent added =            < add , insert >
fluent helpRequested =        < help , { insert , selectCust } >
fluent headerCompleted =      < submit , { insert , undo } >
fluent orderPlaced =      < next , insert } >
fluent confirmed =        < confirm , { insert , undo } >
fluent viewHardDiscs =        < hardDiscs , { RAM , CPUs } >
fluent viewRAM =          < RAM , { hardDiscs , CPUs } >
fluent viewCPUs =         < CPUs , { hardDiscs , RAM } >
fluent never =            < , >
```

Table 2 Showwhen rules for eSuite simulation

Show when	Display
not never	*header to brand page with logos etc (always displayed)*
not companySelected	*list of companies to select from*
companySelected	*navigation link back to company selection page*
companySelected	*name of company selected at top of page*
companySelected **and not** customerSelected	*box to enter customer name or search box to search for customers, plus button for help on this page*
HelpRequested	*page giving instructions on*
Searched	*list of possible customers to select from*
customerSelected	*navigation link back to customer selection page*
customerSelected	*name of customer selected at top of page*
customerSelected **and not** headerCompleted	*form to enter order header data (salesman name, delivery options etc) with submit button*
headerCompleted **and not** confirmed	*overview of the order header data with confirm button*
confirmed **and not** orderPlaced	*table of categories of items for the order*
confirmed **and not** orderPlaced **and** viewCPUs **and not** modifyMode	*list of CPUs to add to order and button to add them*
confirmed **and not** orderPlaced **and** viewCPUs **and** modifyMode	*list of CPUs to add to order and button to add them, plus buttons to remove items from order*
confirmed **and not** orderPlaced **and** viewRAM **and not** modifyMode	*list of RAM chips to add to order and button to add them*
confirmed **and not** orderPlaced **and** viewRAM **and** modifyMode	*list of RAM chips to add to order and button to add them, plus buttons to remove items from order*
confirmed **and not** orderPlaced **and** viewHardDiscs **and not** modifyMode	*list of hard discs to add to order and button to add them*
confirmed **and not** orderPlaced **and** viewHardDiscs **and** modifyMode	*list of hard discs to add to order and button to add them, plus buttons to remove items from order*
confirmed **and not** orderPlaced **and** viewCPUs **and not** modifyMode	*list of CPUs to add to order and button to add them*
orderPlaced	*summary of all order details*

Fig. 11 Refinement of PlaceOrder Goal

```
[](instruction  -> <> companySelected) &&
[](companySelected -> <> orderPlaced) &&
[](companySelected -> <> customerSelected) &&
[](customerSelected -> <> orderPlaced) &&
[](customerSelected -> <> headerCompleted) &&
[](headerCompleted -> <> orderPlaced)
------------------------------------------
[](instruction  -> <> orderPlaced)
```

goals, and what might be changed in order to make it more effective. Suggestions about variations in the order in which activities were performed could be incorporated into the model with a few minutes work. This was aided by the separation of the scenario specification from the abstract state and visualisation representations used in our approach. Once a change had been made (normally in the hMSC) the users could retry the scenarios to see if the new interaction was preferable.

It is more time consuming to introduce entirely new scenarios, and the visualisations for them if during the validation process it is discovered that the decomposition of the overall goal into subgoals was not done correctly, and so it may not be possible to make such a change inside a user-centred design session. However, the effort involved in developing the scenarios and producing an animation is not great. To produce the model of the eSuite application was approximately half a day's work. This is considerably less effort than it would have taken to develop a full prototype version of the application.

10 Discussion and related work

The idea of graphic animation based on a behaviour model is not in itself novel. Many verification tools provide the ability to execute a behaviour model as a way of simulating the system being modelled. The output of this simulation is displayed in the context of the

```
showwhen
  not companySelected
display
  <!— list companies -->
  p “Select company”
  table tr
    td “1”
    td
       link action selectComp
         content “Naftemboriki Newspapers”
         data key=“vcompany”
              value=“Naftemboriki NewsPapers”
    td “2”
    td
       link action selectComp
         content “IBM Hellas”
         data key=“company” value=“IBM Hellas”
    etc…
```

Fig. 12 Snippet of a visualisation rule for eSuite

specification. For example, in SPIN [22] the simulator highlights statements in the Promela specification source as execution proceeds.

Graphical animation in these tools thus refers to animation of some graphical representation of the model specification. This is clearly a useful facility in debugging and understanding models—it is a facility provided in the LTSA which animates LTSs—however, it does not address the problem of communicating in a domain specific way with stakeholders unfamiliar with the modelling formalism.

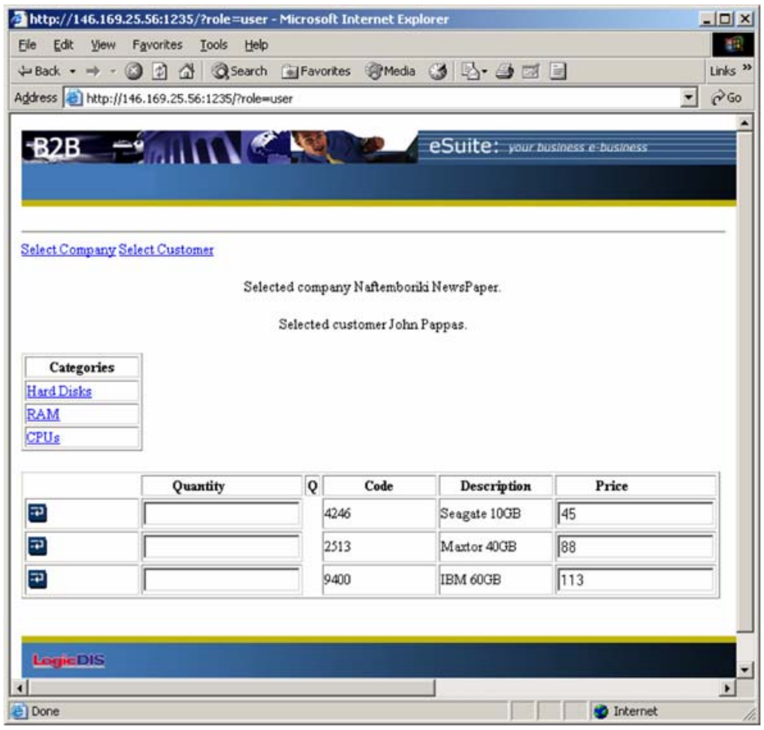
Some initial work on domain specific visualisation is reported by Heitmeyer [18] in the context of the SCR simulator. They use the image of real instrument panels to display the outputs and controls for a simulation of the function of that control panel specified in SCR. Statemate [23] also provides powerful animation capabilities. In both the cases, the visualisation is tied to the structure of the behaviour model and to the occurrence of the last event. The idea of using of fluents as a mechanism to construct visualisations based on abstract system states rather than the concrete states that the model designer may have chosen to specify system behaviour could be built in to existing tools such as SCR, Statemate and others.

In terms of animation based on behaviour provided by scenarios, a noteworthy example is the LSC play-in/play-out tool [24]. The tool requires scenarios to be played in through a mock interface of the system. Once the scenarios are played in, the tool can animate the scenarios through the same mock interface using a similar maximal progress to ours. However, our approach decouples the behavioural specification from the way in which the animation will be visualised. Hence, given one set of scenarios, different visualisations can be tailored according to the particular animation participant.

Note that the LSC scenario notation is more expressive than the one we have adopted. However, our approach to fluent-based animation is independent of the behavioural specification used. The fluents can be used to characterise the system state from a trace, independently of how the trace was generated. Hence, it is possible for our approach to be used in conjunction with other scenario notations with executable semantics, such as LSCs.

Our original work on animation was activity-based animation [25]. There, the goal is to provide smooth

Fig. 13 Screenshot of simulation of eSuite


http://146.169.25.56:1235/?role=user - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://146.169.25.56:1235/?role=user
Go Links
B2B
eSuite: your business e-business
Select Company Select Customer
Categories
Hard Disks
RAM
CPUs
LogicDIS
Done
Internet

Selected company Naftemboriki NewsPaper.

Selected customer John Pappas.

	Quantity	Q	Code	Description	Price
			4246	Seagate 10GB	45
			2513	Maxtor 40GB	88
			9400	IBM 60GB	113

Fig. 14 User-centred design session

animation of the dynamic behaviour of the system between stable, concrete states; a kind of animation well suited for reactive systems. In activity-based animation it is the model that commands the animator to start or stop an animation, for instance the image of a production cell robot arm moving; the user simply changes environment conditions that enable or disable the occurrence of specific actions in the model. The animations discussed in this paper are essentially state-based. The focus is on providing feedback to the user based on the current stable system state (or an abstraction of it). The user triggers actions in the model in response to this feedback. Thus control of animation is almost the opposite of [9]. State-based animation is therefore more suitable for validation of goals formulated as expressions on controllable system states. See Fig. 15 for a depiction of the differences between activity- and state-based animation.

Further recent work on user interface animation [26] relied on a different mechanism for specifying visualisations. In essence, only the events enabled in the current state of the animated model were taken into account to build visualisations of system state. Experience has shown that constructing visualisation on the basis of potential future events is too limiting. The approach frequently leads to animations that are not meaningful because relevant states (from the stakeholders perspective) cannot always be inferred from these events, the history of events that led to the state are typically

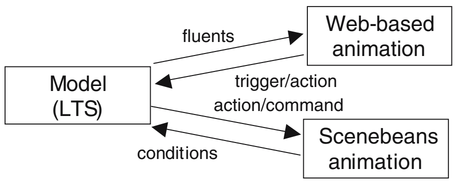


Fig. 15 State-based versus activity-based animation

important. The current fluent-based animation addresses this limitation.

The fact that the animation engine we present is based on web technology makes the animations more suitable in situations in which animation participant interaction drives the animation forward, as in reactive systems. This is due to the fact that the browsers rendering the state of the animation can only pull this information when the participant clicks on a link or presses a button on the web page being visualised. However, a different animation engine that supports pushing information to participants could be used for animating models of a wide range. Indeed, the idea of using of fluents as a mechanism to construct visualisations based on abstract system states rather than the concrete states that the model designer may have chosen to specify system behaviour is suitable for animation of reactive systems independently of the specific behaviour modelling approach. The idea could be built in to existing tools such as SCR, Statemate and others.

The use of scenarios in requirements engineering is certainly a well-developed area (see. e.g. [14, 27]), particularly for requirements validation, elicitation and elaboration. Our work is very much in the spirit of [28] where scenarios are in conjunction with prototyping for goal-oriented requirement validation. However, rather than playing scenarios over a fixed prototype and using probe questions that address system goals, we use behaviour models to drive the walkthrough and use fluents to build the visualisations dynamically. Our work is also in line with the inquiry cycle proposed in [29] where scenarios are used to prompt goal elaboration. In addition, our use of goals as the basis for constructing visualisations is consistent with work on requirements and viewpoints [30, 31].

As pointed out in [6], one of the drawbacks of scenario notations is that they are instance level descriptions. Hence, some generalisation must be done in order to relate them to type-level goals. In our work this is done when defining the fluents in terms of instance specific events. However, this is an area that needs further work. We are currently investigating the use of architecture descriptions in combination with scenarios to improve scenario generalisation.

In [6] a method for inferring goals from scenarios is presented. In essence, what is being inferred is how events change the abstract state of the system. The difficulty resides in knowing what are the relevant state abstractions that should be inferred. In a sense, this is the opposite of what is done in this approach. We take goals that are expressed in terms of abstract system states, and try to find the events that make the system enter and exit these states. These events are what define the fluents used for animation. [6] Also they provide a detailed discussion on the intertwining of scenario and goal-based RE. The focus is mainly on how the elaboration of one can prompt the elaboration of the other. This paper contributes to this intertwining by showing how the combination of both scenarios and goals can be exploited for animation, and hence requirements validation. We believe that visualising how components interact to realise goals helps to facilitate elaboration of requisites and responsibility assignments [3]. Further work is needed to confirm this.

Finally, this work supports the hypothesis that requirements and architectures should be developed hand in hand [32]. Goals are used to clarify the background of the system and explore the high-level design space without necessarily even making a user/machine distinction. In contrast, the notion of system architecture is fundamental in sequence chart notations. By combining both goals and scenarios, architecture comes into play and can inform the elaboration of requirements.

11 Conclusion

A particular novelty of the approach discussed in this paper is the mechanism through which visualisations can be constructed based on abstract system states rather than the concrete states that the model designer may have chosen to specify system behaviour. This allows for greater generality and flexibility, and allows engineers to produce animations that have a concrete relation to the goals that the participating stakeholder has in mind.

As explained in previous sections, our work builds on the goal-based requirements engineering approach of van Lamsweerde. In particular we exploit the fact that goal refinements eventually deliver goals that can be formulated in terms of controllable system states. This is where we introduce fluents to relate these goals to a behavioural specification given in terms of scenarios. Although fluents are the mechanism for relating scenario events with goals, the engineer must still decide which are the events that make each fluent true and false. If the goals are at too high level for a relation with the operational scenarios to be established by inspection, then goal refinement [3] may be helpful to bring goals closer to the operational description. Goal operationalisation [33] could be applied to low-level goals; however, we believe that having an operational description in the form of scenarios, this will not be necessary. In future work we will investigate more rigorous methods for supporting these decisions.

References

1. ITU (2000) Message sequence charts. International Telecommunications Union. Telecommunication Standardisation Sector Recommendation Z.120
2. Object Management Group (2004) Unified modeling language (UML). http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML
3. Lamsweerde AV (2001) Goal-oriented requirements engineering: a guided tour. In: 5th IEEE international symposium on requirements engineering (RE'01), Toronto
4. Dardenne A, van Lamsweerde A, Fickas S (1993) Goal-directed requirements acquisition. Sci Comput Program 20:3–50
5. Feather M (1987) Language support for the specification and development of composite systems. ACM Trans Program Lang Syst 9(2):198–234
6. Lamsweerde AV (1998) Willemet L inferring declarative requirements specifications from operational scenarios. IEEE Trans Softw Eng 24(12):1089–1114
7. Nuseibeh B, Easterbrook S (2000) Requirements engineering: a roadmap. In: International conference on software engineering (ICSE-2000), Limerick
8. Giannakopoulou D, Magee J (2003) Fluent model checking for event-based systems. In: 4th joint meeting of the European software engineering conference and ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE 2003), Helsinki
9. Miller R, Shanahan M (1999) The event calculus in classical logic—alternative axiomatisations. Linkoping Electron Art Comput Inform Sci 4(16):1–27
10. Uchitel S (2003) Elaboration of behaviour models and scenario based specifications using implied scenarios. Ph.D. Thesis in Department of Computing, Imperial College London, London
11. Keller R (1976) Formal verification of parallel programs, Commun ACM 19(7):371–384
12. Hoare CAR (1985) Communicating sequential processes. Prentice Hall, Englewood Cliffs
13. Magee J, Kramer J (1999) Concurrency: state models and Java programs. Wiley, New York
14. Sutcliffe A, Maiden NAM, Minocha S, Manuel D (1998) Supporting scenario-based requirements engineering. IEEE Trans Softw Eng 24(12):1072–1088
15. Jackson M (2004) Seeing more of the world. IEEE Softw 21(6):83–85
16. Jackson M (1995) Software requirements and specifications—a lexicon of practice, principles and prejudices. ACM Press, Addison-Wesley, New York, Reading
17. Magee et al (2003) The LTSA site, www.doc.ic.ac.uk/ltsa
18. Heitmeyer C, Kirby C, Labaw B (1997) The SCR method for formally specifying, verifying and validating requirements: tool support. In: 19th International conference on software engineering (ICSE'97), Boston
19. Halbwachs N Caspi P, Raymond P, Pilaud D (1991) The synchronous data-flow programming language LUSTRE. Proc IEEE, 79:1305–1320
20. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152
21. Darimont R, Lamsweerde AV (1996) Formal refinement patterns for goal-driven requirements elaboration. In: 4th ACM symposium on the foundations of software engineering, San Fransisco
22. Holzmann GJ, Peled D (1996) The state of spin. In: CAV'96
23. Harel D, Lachover H, Naamad A, Pnueli A, Politi M, Sheman R, Shtul-Trauring A, Trakhtenbrot M (1990) STATEMATE: a working environment for the development of complex reactive systems. IEEE Trans Softw Eng 16:403–414
24. Harel D, Marelly R (2003) Come, let's play: scenario-based programming using LSCs and the play-engine. Springer, Berlin, Heidelberg, New York
25. Magee J, Kramer J, Giannakopoulou D, Pryce N (2000) Graphical animation of behavior models. In: 22nd International conference on software engineering (ICSE'00), Limerick
26. Magee J, Uchitel S, Chatley R, Kramer J (2003) Visual methods for Web application design. In: Tech note at the symposium on visual and multimedia software engineering, IEEE symposia on human-centric computing languages and environments (HCC), Auckland
27. CREWS (1999) Cooperative requirements engineering with scenarios, http://Sunsite.Informatik.RWTH-Aachen.DE/CREWS

28. Sutcliffe A (1997) A technique combination approach to requirements engineering. In: 3rd IEEE international symposium on requirements engineering, Los Alamitos
29. Potts C, Takahashi K, Anton AI (1994) Inquiry-based requirements analysis. IEEE Softw 11(2):21–32
30. Leite J, Freeman PA (1991) Requirements validation through viewpoint resolution. IEEE Trans Softw Eng 12(12):1253–1269
31. Nuseibeh B, Kramer J, Finkelstein A (1994) A framework for expressing the relationships between multiple views in requirements specification. Trans Softw Eng 20(10):760–773
32. Nuseibeh B (2001) Weaving together requirements and architecture. IEEE Comput 34(3):115–117
33. Letier E (2001) Reasoning about agents in goal-oriented requirements engineering. PhD Thesis, Université Catholique de Louvain